模拟试卷 7

单选

1. 下列哪一叙述是正确的（）
   A) abstract 修饰符可修饰字段、方法和类
   B) 抽象方法的 body 部分必须用一对大括号{}包住
   C) 声明抽象方法，大括号可有可无
   D) 声明抽象方法不可写出大括号
   解析:
   abstract 修饰符不能修饰字段;
   一个类如果用 abstract 关键字声明的话,那么必须有抽象方法,该抽象方法也可以是实现的接口中的???
2. 构造函数何时被调用
   A) 类定义时
   B) 创建对象时
   C) 调用对象方法时
   D) 使用对象的变量时
3. 怎样更改一个文件的权限设置（）
   A) attrib
   B) chmod
   C) change
   D) file
4. 如果希望监听 TCP 端口 9000，应该怎样创建 socket()
   A) new Socket("localhost",9000)
   B) new ServerSocket(9000)
   C) new Socket(9000)
   D) new ServerSocket("localhost",9000)
5. 下面关于垃圾收集的说法正确的是（）
   A) 一旦一个对象成为垃圾，就立刻被收集掉
   B) 对象空间被收集掉之后，会执行该对象的 finalize 方法
   C) finalize 方法和 C++的析构函数是完全一回事情
   D) 一个对象成为垃圾是因为不再有引用指着它，但是线程并非如此
6. 下面哪个不是 java 关键字
   A) integer
   B) double
   C) float
   D) default
7. 下列哪个 linux 命令显示一页内容
   A) pause
   B) cat
   C) more
   D) grep

8. 以下关于正反向代理错误的是（）
   A) 正向代理是一个位于客户端和原始服务器之间的服务代理，为了从原理服务器取得内容，客户端向代理发送请求并指定目标（原始服务器）,然后代理向原理服务器转交请求并将获得的内容返回客户端
   B) 正向代理客户端必须要进行一些特别的设置才能使用正向代理
   C) 客户端反向代理，命名空间中的内容发送普通请求，接着反向代理将判断向何处（原理服务器）转发请求，并将获得的内容返回客户端；
   D) 反向代理客户端必须要进行一些特别的设置才能使用反向代理
9. 为了对某序列进行二分查找，则要求其
   A) 可以是乱序
   B) 必须已排序
   C) 可以是乱序也可以是已排序
   D) 必须转化为二叉树
10. 以下说法正确的是（）
   A) 外键必须和相应的主键同名
   B) 外键值不可以为空
   C) 外键可以和相应的主键不同名，只要定义在相同的域上即可
   D) 外键的取值只允许等于所参照关系中的某个主键值
11. ToudaDFS 中,临时文件保存()天
   A) 7
   B) 15
   C) 30
   D) 60
12. 关于 touda 渠道服务，描述错误的是（）
   A) 渠道服务处理报文打解包功能
   B) 渠道服务可以对报文的输入进行校验
   C) 渠道服务可以配置交易事务
   D) 渠道服务可以控制交易路由
13. 关于 touda 项目的说法，错误的是（）
   A) 创建项目时，可根据实际情况，选择创建架构设计构建包
   B) 项目中必须包含业务建模构建包
   C) 项目中必须包含服务定制模块
   D) 本地进行项目开发时，需要引入 function 工程
14. 2n 个数中的最大值和最小值，最少的比较次数是（）
   A) 4n/3
   B) 2n-2
   C) 3n-2
   D) 3n/2
15. 被提及的 Dos 攻击的是以下的行为（）
   A) 侵入目标服务器，获取重要数据
   B) 采用穷举的方式获得登录账号
   C) 发送无效的请求，使得正确的请求无法被响应
   D) 利用微软 DOS 从操作系统图的各种漏洞达到攻击目的
16. 关于 touda 服务开发，说法错误的有（）

A) Touda 日志打印允许第三方日志 api
B) 渠道服务命名规范为渠道类型（J/X）+_+交易码，不得超过 20 字符
C) 渠道类型简称 J 代表 json 格式报文，X 代表 SOAP 格式报文
D) Touda 服务在引用第三方 jar 包时优先考虑 toudaServer 已经依赖的 jar 包

17. 关于 Touda 服务注册中心,下列说法正确的是
A) 非 Touda 服务调用,无法 Touda 服务注册中心进行服务订阅
B) Touda 服务实际调用时,请求统一由 Touda 服务注册中心进行转发
C) Touda 提供的负载均衡功能是在服务调用方完成的
D) Touda 服务调用方在每次调用时,从 Touda 注册中心拉取服务订阅列表信息

18. 关于 function 的功能,说法错误的有
A) 提供渠道服务的服务注册功能
B) 提供主机服务的服务订阅功能
C) 提供服务器参数动态配置的功能
D) 提供 Touda 应用的启动停止功能

19. 关于 touda 交易流，下列描述错误的是（）
A) 交易流中可以直接调用本地处理
B) 交易流中可以直接调用运算逻辑
C) 交易流中可以直接调用渠道服务
D) 交易流中可以直接调用主机服务

20. ps 使用什么参数查看全部进程
A) a
B) b
C) u
D) x

21. 关于 touda 渠道服务说法，错误的是（）
A) touda 服务可以创建 HTTP 渠道
B) touda 服务可以创建 TCP 渠道
C) touda 服务不可以自定义渠道
D) touda 服务可以创建 MQ 渠道

22. 关于 touda 通道的说法，错误的是（）
A) touda 服务可以创建 HTTP 通道
B) touda 服务可以创建 TCP 通道
C) touda 服务可以自定义通道
D) 通过 touda 通道调用外部服务不可以通过 touda 服务从服务注册中心获取服务列表

23. 关于 logback 日志打印级别，正确的是（）
A) TRACE<DEBUG<INFO<WARN<ERROR
B) DEBUG<TRACE<INFO<WARN<ERROR
C) TRACE<DEBUG<WARN<INFO<ERROR
D) TRACE<DEBUG<INFO<ERROR<WARN

24. 关于配置中心,下列说法错误的是()
A) 配置中心支持集群发布
B) 配置中心同时支持云上和云下
C) 配置中心不支持实时生效
D) 配置中心支持和查看示例列表(使用指定配置的服务器信息)

25. 关于云上应用，下列说法错误的是（）
    A) 云上应用日志不允许输出到本地，查看日志需要登录统一日志平台查看
    B) 新增的微应用，需要走双数流程申请，取得合法应用名
    C) tomcat 应用上云配置项无需在配置中心进行管理
    D) touda 应用上云，渠道暴露端口必须为 8080

多选

1. 关于 java 队列正确的是
    A) LinkedBlockingQueue 是可选有界队列，不允许有 null 值
    B) LinkedBlockingQueue 和 PriorityQueue 都是线程安全的
    C) PriorotyQueue 是无界队列，不允许有 null 值，入队出队时间复杂度为 0（log(n)）
    D) PriorityQueue 和 ConCurrentLinkedQueue 都遵循 FIFO 原则
2. 以下哪一项属于持续集成中属于每日构建、而不是持续构建的任务
    A. 自动上传开发环境、保持环境和源码同步
    B. 时时扫描代码变更
    C. 自动触发编译部署任务
    D. 如果有编译失败自动邮件通知项目组
3. 配置管理架构流程不涉及以下哪项角色
    A．IT 系统管理团队
    B．业务人员
    C．IT 开发经理
    D．IT 配置管理团队
4. 下面哪些是死锁发生的必要条件
    A) 互斥条件
    B) 请求和保持
    C) 不可剥夺
    D) 循环等待
5. jvm 中垃圾回收分为 scanvenge gc 和 full GC 其中 full GC 触发的条件可能有哪些
    A) 栈空间满
    B) 年轻代空间满
    C) 老年代满
    D) 持久代满
    E) System.gc()
6. 以下说法正确的是()
    A) 直连时,touda.properties 中的 connect_flag 的值为 1
    B) 在 UAT 测试时,必须使用非直连方式
    C) 直连是直接请求在 spring-proxy.xml 中配置的 HTTP URL
    D) 非直连这需要根据在 spring-proxy.xml 中配置的服务名到注册中心获取服务 URL 列表
7. 在创建新项目时,以下说法正确的是()
    A) 创建项目时,首先需要检查 PrimetonEOS 是否设置正确
    B) touda 项目使用的 JRE 版本必须设置为 1.7 以上

C) 创建新项目时,"业务建模"和"服务定制"是 2 个需要新增的构建包

D) Touda 工程中的所有配置项必须在 touda.properties 中进行统一管理

8. 以下关于多线程说法 正确的是()

A) start()和 run()方法都可以启动线程

B) CyclicBarrier 和 CountDownLatch 都能让一组线程等待其他线程

C) Callable 的 call()方法可以返回值和抛出异常

D) 新建的线程,start()之后立即运行

9. 以下说法错误的是()

A) 数组是一种对象

B) 数组是一种原生类

C) int number[] = {23,33,43,46,60}

D) 数据的大小可以任意改变

10. 下面哪几个描述是正确的（）

A) 默认构造器初始化方法变更

B) 默认构造器有和它所在类相同的访问修饰词

C) 默认构造器调用其父类的无参构造器

D) 如果一个类没有无参构造器，编译器会为它创建一个默认构造器

E) 只有当一个类没有任何构造器时，编译器会为它创建一个默认构造器

编程

1 适配器式是 java 常见的设计模式之一，请写一段代码实现适配器模式

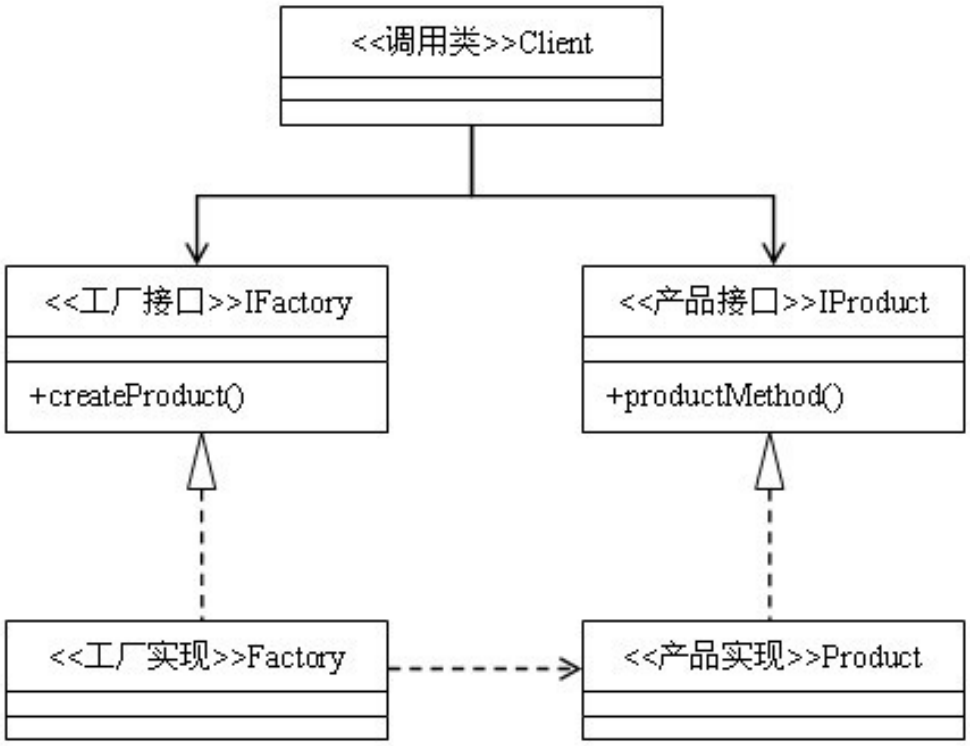
具体实现参考：http://www.runoob.com/design-pattern/adapter-pattern.html

抽象工厂模式

定义：为创建一组相关或相互依赖的对象提供一个接口，而且无需指定他们的具体类。

类型：创建类模式

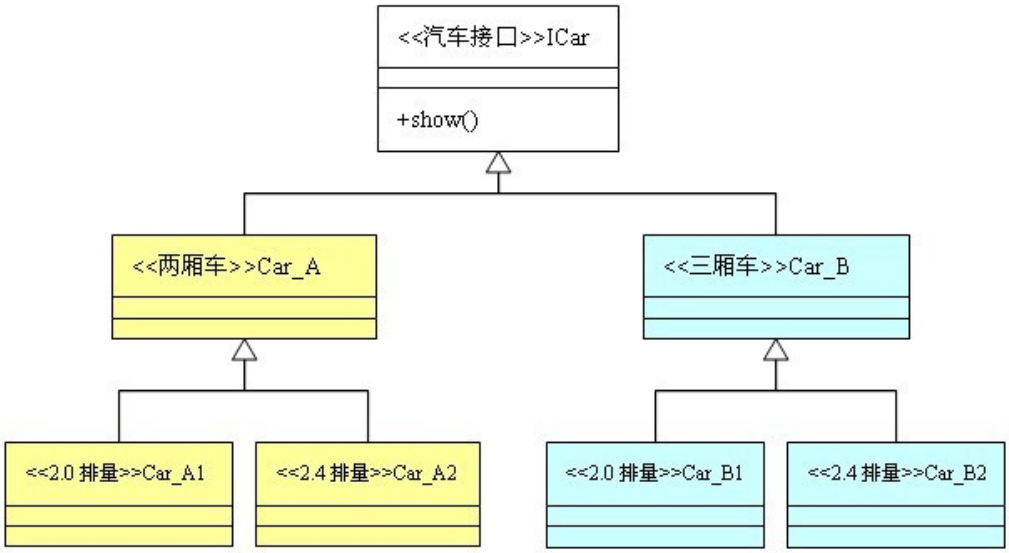
类图：



抽象工厂模式与工厂方法模式的区别

抽象工厂模式是工厂方法模式的升级版本，他用来创建一组相关或者相互依赖的对象。他与工厂方法模式的区别就在于，工厂方法模式针对的是一个产品等级结构；而抽象工厂模式则是针对的多个产品等级结构。在编程中，通常一个产品结构，表现为一个接口或者抽象类，也就是说，工厂方法模式提供的所有产品都是衍生自同一个接口或抽象类，而抽象工厂模式所提供的产品则是衍生自不同的接口或抽象类。

在抽象工厂模式中，有一个**产品族**的概念：所谓的产品族，是指**位于不同产品等级结构中功能相关联的产品组成的家族**。抽象工厂模式所提供的一系列产品就组成一个产品族；而工厂方法提供的一系列产品称为一个等级结构。我们依然拿生产汽车的例子来说明他们之间的区别。



在上面的类图中，两厢车和三厢车称为两个不同的等级结构；而 2.0 排量车和 2.4 排量车则称为两个不同的产品族。再具体一点，2.0 排量两厢车和 2.4 排量两厢车属于同一个等级结构，2.0 排量三厢车和 2.4 排量三厢车属于另一个等级结构；而 2.0 排量两厢车和 2.0 排量三厢车属于同一个产品族，2.4 排量两厢车和 2.4 排量三厢车属于另一个产品族。

明白了等级结构和产品族的概念，就理解工厂方法模式和抽象工厂模式的区别了，如果工厂的产品全部属于同一个等级结构，则属于工厂方法模式；如果工厂的产品来自多个等级结构，则属于抽象工厂模式。在本例中，如果一个工厂模式提供 2.0 排量两厢车和 2.4 排量两厢车，那么他属于工厂方法模式；如果一个工厂模式是提供 2.4 排量两厢车和 2.4 排量三厢车两个产品，那么这个工厂模式就是抽象工厂模式，因为他提供的产品是分属两个不同的等级结构。当然，如果一个工厂提供全部四种车型的产品，因为产品分属两个等级结构，他当然也属于抽象工厂模式了。

抽象工厂模式代码

```
interface IProduct1 {
    public void show();
}
interface IProduct2 {
    public void show();
}

class Product1 implements IProduct1 {
    public void show() {
        System.out.println("这是 1 型产品");
    }
```

```
}
class Product2 implements IProduct2 {
    public void show() {
        System.out.println("这是 2 型产品");
    }
}

interface IFactory {
    public IProduct1 createProduct1();
    public IProduct2 createProduct2();
}
class Factory implements IFactory{
    public IProduct1 createProduct1() {
        return new Product1();
    }
    public IProduct2 createProduct2() {
        return new Product2();
    }
}

public class Client {
    public static void main(String[] args){
        IFactory factory = new Factory();
        factory.createProduct1().show();
        factory.createProduct2().show();
    }
}
```

抽象工厂模式的优点

抽象工厂模式除了具有工厂方法模式的优点外，最主要的优点就是可以在类的内部对产品族进行约束。所谓的产品族，一般或多或少的都存在一定的关联，抽象工厂模式就可以在类内部对产品族的关联关系进行定义和描述，而不必专门引入一个新的类来进行管理。

抽象工厂模式的缺点

产品族的扩展将是一件十分费力的事情，假如产品族中需要增加一个新的产品，则几乎所有的工厂类都需要进行修改。所以使用抽象工厂模式时，对产品等级结构的划分是非常重要的。

适用场景

当需要创建的对象是一系列相互关联或相互依赖的产品族时，便可以使用抽象工厂模式。说的更明白一点，就是一个继承体系中，如果存在着多个等级结构（即存在着多个抽象类），并且分属各个等级结构中的实现类之间存在着一定的关联或者约束，就可以使用抽

象工厂模式。假如各个等级结构中的实现类之间不存在关联或约束，则使用多个独立的工厂来对产品进行创建，则更合适一点。

总结

无论是简单工厂模式，工厂方法模式，还是抽象工厂模式，他们都属于工厂模式，在形式和特点上也是极为相似的，他们的最终目的都是为了解耦。在使用时，我们不必去在意这个模式到底工厂方法模式还是抽象工厂模式，因为他们之间的演变常常是令人琢磨不透的。经常你会发现，明明使用的工厂方法模式，当新需求来临，稍加修改，加入了一个新方法后，由于类中的产品构成了不同等级结构中的产品族，它就变成抽象工厂模式了；而对于抽象工厂模式，当减少一个方法使的提供的产品不再构成产品族之后，它就演变成了工厂方法模式。

所以，在使用工厂模式时，只需要关心降低耦合度的目的是否达到了。

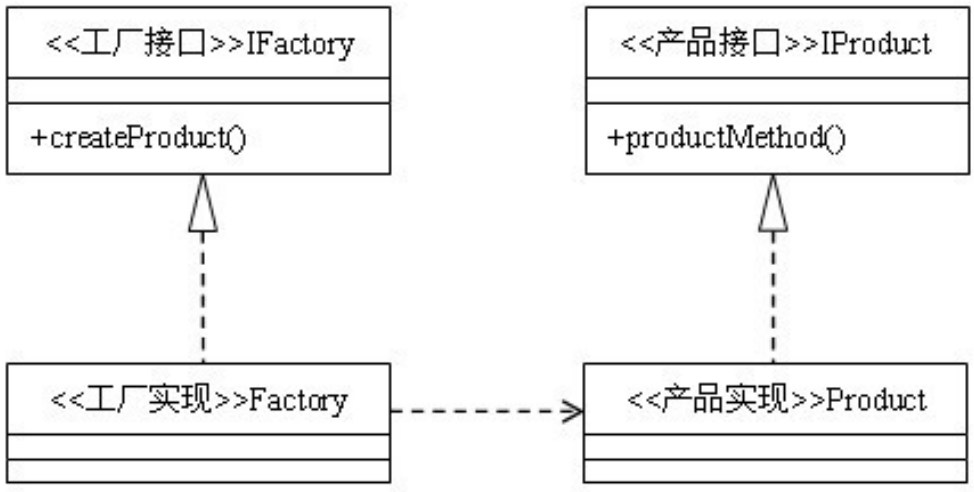
工厂方法模式

定义：定义一个用于创建对象的接口，让子类决定实例化哪一个类，工厂方法使一个类的实例化延迟到其子类。

类型：创建类模式

类图：

!



工厂方法模式代码

```
    interface IProduct {
        public void productMethod();
```

```
}

class Product implements IProduct {
    public void productMethod() {
        System.out.println("产品");
    }
}

interface IFactory {
    public IProduct createProduct();
}

class Factory implements IFactory {
    public IProduct createProduct() {
        return new Product();
    }
}

public class Client {
    public static void main(String[] args) {
        IFactory factory = new Factory();
        IProduct prodect = factory.createProduct();
        prodect.productMethod();
    }
}
```

工厂模式：

首先需要说一下工厂模式。工厂模式根据抽象程度的不同分为三种：简单工厂模式（也叫静态工厂模式）、本文所讲述的**工厂方法模式**、以及抽象工厂模式。工厂模式是编程中经常用到的一种模式。它的主要优点有：

- 可以使代码结构清晰，有效地封装变化。在编程中，产品类的实例化有时候是比较复杂和多变的，通过工厂模式，将产品的实例化封装起来，使得调用者根本无需关心产品的实例化过程，只需依赖工厂即可得到自己想要的产品。
- 对调用者屏蔽具体的产品类。如果使用工厂模式，调用者只关心产品的接口就可以了，至于具体的实现，调用者根本无需关心。即使变更了具体的实现，对调用者来说没有任何影响。
- 降低耦合度。产品类的实例化通常来说是很复杂的，它需要依赖很多的类，而这些类对于调用者来说根本无需知道，如果使用了工厂方法，我们需要做的仅仅是实例化好产品类，然后交给调用者使用。对调用者来说，产品所依赖的类都是透明的。

工厂方法模式：

通过工厂方法模式的类图可以看到，工厂方法模式有四个要素：

- 工厂接口。工厂接口是工厂方法模式的核心，与调用者直接交互用来提供产品。在实际编程中，有时候也会使用一个抽象类来作为与调用者交互的接口，其本质上是一样的。

- 工厂实现。在编程中，工厂实现决定如何实例化产品，是实现扩展的途径，需要有多少种产品，就需要有多少个具体的工厂实现。
- 产品接口。产品接口的主要目的是定义产品的规范，所有的产品实现都必须遵循产品接口定义的规范。产品接口是调用者最为关心的，产品接口定义的优劣直接决定了调用者代码的稳定性。同样，产品接口也可以用抽象类来代替，但要注意最好不要违反里氏替换原则。
- 产品实现。实现产品接口的具体类，决定了产品在客户端中的具体行为。

前文提到的简单工厂模式跟工厂方法模式极为相似，区别是：简单工厂只有三个要素，他没有工厂接口，并且得到产品的方法一般是静态的。因为没有工厂接口，所以在工厂实现的扩展性方面稍弱，可以算所工厂方法模式的简化版，关于简单工厂模式，在此一笔带过。

适用场景：

不管是简单工厂模式，工厂方法模式还是抽象工厂模式，他们具有类似的特性，所以他们的适用场景也是类似的。

首先，作为一种创建类模式，在任何需要生成**复杂对象**的地方，都可以使用工厂方法模式。有一点需要注意的地方就是复杂对象适合使用工厂模式，而简单对象，特别是只需要通过 new 就可以完成创建的对象，无需使用工厂模式。如果使用工厂模式，就需要引入一个工厂类，会增加系统的复杂度。
其次，工厂模式是一种典型的解耦模式，迪米特法则在工厂模式中表现的尤为明显。假如调用者自己组装产品需要增加依赖关系时，可以考虑使用工厂模式。将会大大降低对象之间的耦合度。

再次，由于工厂模式是依靠抽象架构的，它把实例化产品的任务交由实现类完成，扩展性比较好。也就是说，当需要系统有比较好的扩展性时，可以考虑工厂模式，不同的产品用不同的实现工厂来组装。

典型应用

要说明工厂模式的优点，可能没有比组装汽车更合适的例子了。场景是这样的：汽车由发动机、轮、底盘组成，现在需要组装一辆车交给调用者。假如不使用工厂模式，代码如下：

```
    class Engine {
        public void getStyle(){
            System.out.println("这是汽车的发动机");
        }
    }
    class Underpan {
        public void getStyle(){
            System.out.println("这是汽车的底盘");
```

```
        }
    }
    class Wheel {
        public void getStyle(){
            System.out.println("这是汽车的轮胎");
        }
    }
    public class Client {
        public static void main(String[] args) {
            Engine engine = new Engine();
            Underpan underpan = new Underpan();
            Wheel wheel = new Wheel();
            ICar car = new Car(underpan, wheel, engine);
            car.show();
        }
    }
```

可以看到，调用者为了组装汽车还需要另外实例化发动机、底盘和轮胎，而这些汽车的组件是与调用者无关的，严重违反了迪米特法则，耦合度太高。并且非常不利于扩展。另外，本例中发动机、底盘和轮胎还是比较具体的，在实际应用中，可能这些产品的组件也都是抽象的，调用者根本不知道怎样组装产品。假如使用工厂方法的话，整个架构就显得清晰了许多。

```
    interface IFactory {
        public ICar createCar();
    }
    class Factory implements IFactory {
        public ICar createCar() {
            Engine engine = new Engine();
            Underpan underpan = new Underpan();
            Wheel wheel = new Wheel();
            ICar car = new Car(underpan, wheel, engine);
            return car;
        }
    }
    public class Client {
        public static void main(String[] args) {
            IFactory factory = new Factory();
            ICar car = factory.createCar();
            car.show();
        }
    }
```

使用工厂方法后，调用端的耦合度大大降低了。并且对于工厂来说，是可以扩展的，以后如果想组装其他的汽车，只需要再增加一个工厂类的实现就可以。无论是灵活性还是稳定性都得到了极大的提高。

单例模式

定义：确保一个类只有一个实例，而且自行实例化并向整个系统提供这个实例。
类型：创建类模式
类图：

类图知识点：

1.类图分为三部分，依次是类名、属性、方法

2.以<<开头和以>>结尾的为注释信息

3.修饰符+代表 public，-代表 private，#代表 protected，什么都没有代表包可见。

4.带下划线的属性或方法代表是静态的。

5.对类图中对象的关系不熟悉的朋友可以参考文章：设计模式中类的关系。

单例模式应该是 23 种设计模式中最简单的一种模式了。它有以下几个要素：

- 私有的构造方法
- 指向自己实例的私有静态引用
- 以自己实例为返回值的静态的公有的方法

单例模式根据实例化对象时机的不同分为两种：一种是饿汉式单例，一种是懒汉式单例。饿汉式单例在单例类被加载时候，就实例化一个对象交给自己的引用；而懒汉式在调用取得实例方法的时候才会实例化对象。代码如下：

饿汉式单例

```
    public class Singleton {
```

```
        private static Singleton singleton = new Singleton();
        private Singleton(){}
        public static Singleton getInstance(){
            return singleton;
        }
    }
```

懒汉式单例

```
    public class Singleton {
        private static Singleton singleton;
        private Singleton(){}

        public static synchronized Singleton getInstance(){
            if(singleton==null){
                singleton = new Singleton();
            }
            return singleton;
        }
    }
```

单例模式的优点：

- 在内存中只有一个对象，节省内存空间。
- 避免频繁的创建销毁对象，可以提高性能。
- 避免对共享资源的多重占用。
- 可以全局访问。

适用场景：由于单例模式的以上优点，所以是编程中用的比较多的一种设计模式。我总结了一下我所知道的适合使用单例模式的场景：

- 需要频繁实例化然后销毁的对象。
- 创建对象时耗时过多或者耗资源过多，但又经常用到的对象。
- 有状态的工具类对象。
- 频繁访问数据库或文件的对象。
- 以及其他我没用过的所有要求只有一个对象的场景。

单例模式注意事项：

- 只能使用单例类提供的方法得到单例对象，不要使用反射，否则将会实例化一个新对象。
- 不要做断开单例类对象与类中静态引用的危险操作。
- 多线程使用单例使用共享资源时，注意线程安全问题。

关于 java 中单例模式的一些争议：

单例模式的对象长时间不用会被 jvm 垃圾收集器收集吗

看到不少资料中说：如果一个单例对象在内存中长久不用，会被 jvm 认为是一个垃圾，在执行垃圾收集的时候会被清理掉。对此这个说法，笔者持怀疑态度，笔者本人的观点是：

在 hotspot 虚拟机 1.6 版本中，除非人为地断开单例中静态引用到单例对象的联接，否则 jvm 垃圾收集器是不会回收单例对象的。

对于这个争议，笔者单独写了一篇文章进行讨论，如果您有不同的观点或者有过这方面的经历请进入文章单例模式讨论篇：单例模式与垃圾收集参与讨论。

在一个 jvm 中会出现多个单例吗

在分布式系统、多个类加载器、以及序列化的的情况下，会产生多个单例，这一点是无庸置疑的。那么在同一个 jvm 中，会不会产生单例呢？使用单例提供的 getInstance()方法只能得到同一个单例，除非是使用反射方式，将会得到新的单例。代码如下

```
Class c = Class.forName(Singleton.class.getName());
Constructor ct = c.getDeclaredConstructor();
ct.setAccessible(true);
Singleton singleton = (Singleton)ct.newInstance();
```

这样，每次运行都会产生新的单例对象。所以运用单例模式时，一定注意不要使用反射产生新的单例对象。

懒汉式单例线程安全吗

主要是网上的一些说法，懒汉式的单例模式是线程不安全的，即使是在实例化对象的方法上加 **synchronized** 关键字，也依然是危险的，但是笔者经过编码测试，发现加 **synchronized** 关键字修饰后，虽然对性能有部分影响，但是却是线程安全的，并不会产生实例化多个对象的情况。

单例模式只有饿汉式和懒汉式两种吗

饿汉式单例和懒汉式单例只是两种比较主流和常用的单例模式方法，从理论上讲，任何可以实现一个类只有一个实例的设计模式，都可以称为单例模式。

单例类可以被继承吗

饿汉式单例和懒汉式单例由于构造方法是 private 的，所以他们都是不可继承的，但是其他很多单例模式是可以继承的，例如登记式单例。

饿汉式单例好还是懒汉式单例好

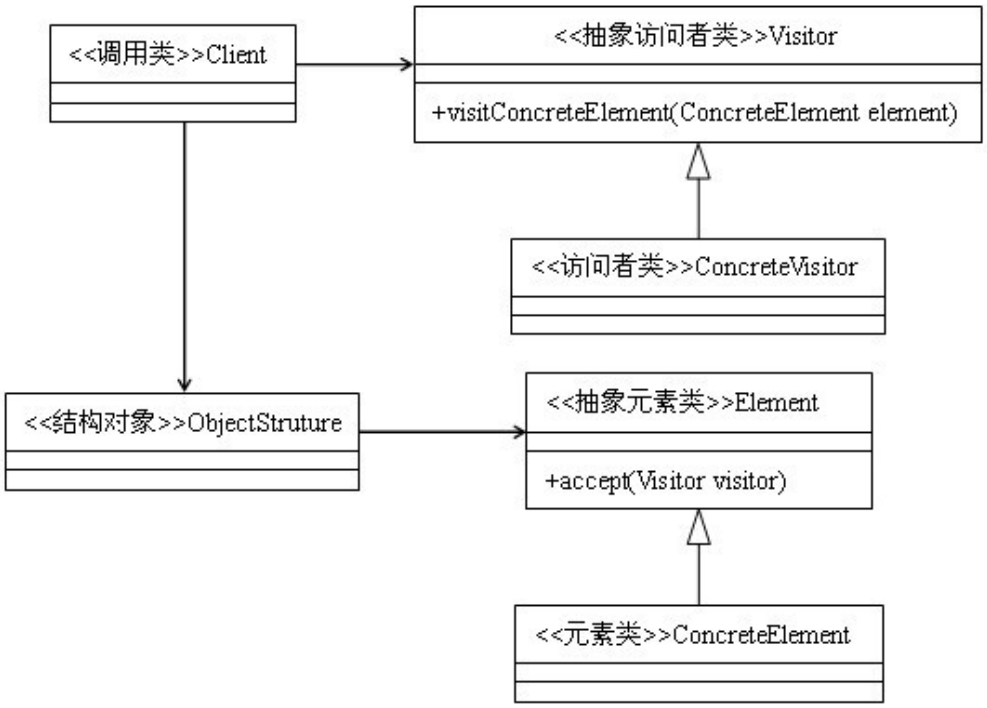
在 java 中，饿汉式单例要优于懒汉式单例。C++中则一般使用懒汉式单例。

访问者模式

定义：封装某些作用于某种数据结构中各元素的操作，它可以在不改变数据结构的前提下定义作用于这些元素的新的操作。

类型：行为类模式

类图：



访问者模式可能是行为类模式中最复杂的一种模式了，但是这不能成为我们不去掌握它的理由。我们首先来看一个简单的例子，代码如下：

```
class A {
    public void method1(){
        System.out.println("我是 A");
    }

    public void method2(B b){
        b.showA(this);
    }
}

class B {
    public void showA(A a){
        a.method1();
    }
}
```

我们主要来看一下在类 A 中，方法 method1 和方法 method2 的区别在哪里，方法 method1 很简单，就是打印出一句"我是 A"；方法 method2 稍微复杂一点，使用类 B 作为参数，并调用类 B 的 showA 方法。再来看一下类 B 的 showA 方法，showA 方法使用类 A 作为参数，然后调用类 A 的 method1 方法，可以看到，method2 方法绕来绕

去，无非就是调用了一下自己的 method1 方法而已，它的运行结果应该也是"我是 A"，分析完之后，我们来运行一下这两个方法，并看一下运行结果：

```
public class Test {
    public static void main(String[] args){
        A a = new A();
        a.method1();
        a.method2(new B());
    }
}
```

运行结果为：

我是 A

我是 A

看懂了这个例子，就理解了访问者模式的 90%，在例子中，对于类 A 来说，类 B 就是一个访问者。但是这个例子并不是访问者模式的全部，虽然直观，但是它的可扩展性比较差，下面我们就来说一下访问者模式的通用实现，通过类图可以看到，在访问者模式中，主要包括下面几个角色：

- **抽象访问者：**抽象类或者接口，声明访问者可以访问哪些元素，具体到程序中就是 visit 方法中的参数定义哪些对象是可以被访问的。
- **访问者：**实现抽象访问者所声明的方法，它影响到访问者访问到一个类后该干什么，要做什么事情。
- **抽象元素类：**接口或者抽象类，声明接受哪一类访问者访问，程序上是通过 accept 方法中的参数来定义的。抽象元素一般有两类方法，一部分是本身的业务逻辑，另外就是允许接收哪类访问者来访问。
- **元素类：**实现抽象元素类所声明的 accept 方法，通常都是 visitor.visit(this)，基本上已经形成一种定式了。
- **结构对象：**一个元素的容器，一般包含一个容纳多个不同类、不同接口的容器，如 List、Set、Map 等，在项目中一般很少抽象出这个角色。

  访问者模式的通用代码实现

```
abstract class Element {
    public abstract void accept(IVisitor visitor);
    public abstract void doSomething();
}

interface IVisitor {
    public void visit(ConcreteElement1 el1);
    public void visit(ConcreteElement2 el2);
}
```

```
class ConcreteElement1 extends Element {
    public void doSomething(){
        System.out.println("这是元素 1");
    }

    public void accept(IVisitor visitor) {
        visitor.visit(this);
    }
}

class ConcreteElement2 extends Element {
    public void doSomething(){
        System.out.println("这是元素 2");
    }

    public void accept(IVisitor visitor) {
        visitor.visit(this);
    }
}
class Visitor implements IVisitor {

    public void visit(ConcreteElement1 el1) {
        el1.doSomething();
    }

    public void visit(ConcreteElement2 el2) {
        el2.doSomething();
    }
}

class ObjectStruture {
    public static List getList(){
        List list = new ArrayList();
        Random ran = new Random();
        for(int i=0; i<10; i++){
            int a = ran.nextInt(100);
            if(a>50){
                list.add(new ConcreteElement1());
            }else{
                list.add(new ConcreteElement2());
            }
        }
        return list;
    }
```

```
    }

    public class Client {
        public static void main(String[] args){
            List list = ObjectStruture.getList();
            for(Element e: list){
                e.accept(new Visitor());
            }
        }
    }
```

访问者模式的优点

- **符合单一职责原则：**凡是适用访问者模式的场景中，元素类中需要封装在访问者中的操作必定是与元素类本身关系不大且是易变的操作，使用访问者模式一方面符合单一职责原则，另一方面，因为被封装的操作通常来说都是易变的，所以当发生变化时，就可以在不改变元素类本身的前提下，实现对变化部分的扩展。
- **扩展性良好：**元素类可以通过接受不同的访问者来实现对不同操作的扩展。

访问者模式的适用场景

假如一个对象中存在着一些与本对象不相干（或者关系较弱）的操作，为了避免这些操作污染这个对象，则可以使用访问者模式来把这些操作封装到访问者中去。

假如一组对象中，存在着相似的操作，为了避免出现大量重复的代码，也可以将这些重复的操作封装到访问者中去。

但是，访问者模式并不是那么完美，它也有着致命的缺陷：增加新的元素类比较困难。通过访问者模式的代码可以看到，在访问者类中，每一个元素类都有它对应的处理方法，也就是说，每增加一个元素类都需要修改访问者类（也包括访问者类的子类或者实现类），修改起来相当麻烦。也就是说，在元素类数目不确定的情况下，应该慎用访问者模式。所以，访问者模式比较适用于对已有功能的重构，比如说，一个项目的基本功能已经确定下来，元素类的数据已经基本确定下来不会变了，会变的只是这些元素内的相关操作，这时候，我们可以使用访问者模式对原有的代码进行重构一遍，这样一来，就可以在不修改各个元素类的情况下，对原有功能进行修改。

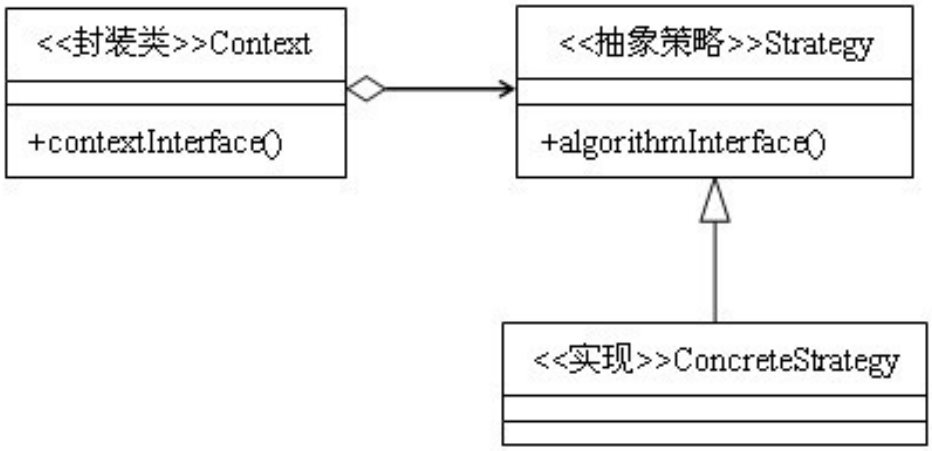
总结

正如《设计模式》的作者 GoF 对访问者模式的描述：大多数情况下，你并需要使用访问者模式，但是当你一旦需要使用它时，那你就是真的需要它了。当然这只是针对真正的大牛而言。在现实情况下（至少是我所处的环境当中），很多人往往沉迷于设计模式，他们使用一种设计模式时，从来不去认真考虑所使用的模式是否适合这种场景，而往往只是想展示一下自己对面向对象设计的驾驭能力。编程时有这种心理，往往会发生滥用设计模式的

情况。所以，在学习设计模式时，一定要理解模式的适用性。必须做到使用一种模式是因为了解它的优点，不使用一种模式是因为了解它的弊端；而不是使用一种模式是因为不了解它的弊端，不使用一种模式是因为不了解它的优点。

策略模式

定义：定义一组算法，将每个算法都封装起来，并且使他们之间可以互换。
类型：行为类模式
类图：



策略模式是对算法的封装，把一系列的算法分别封装到对应的类中，并且这些类实现相同的接口，相互之间可以替换。在前面说过的行为类模式中，有一种模式也是关注对算法的封装——模版方法模式，对照类图可以看到，策略模式与模版方法模式的区别仅仅是多了一个单独的封装类 Context，它与模版方法模式的区别在于：在模版方法模式中，调用算法的主体在抽象的父类中，而在策略模式中，调用算法的主体则是封装到了封装类 Context 中，抽象策略 Strategy 一般是一个接口，目的只是为了定义规范，里面一般不包含逻辑。其实，这只是通用实现，而在实际编程中，因为各个具体策略实现类之间难免存在一些相同的逻辑，为了避免重复的代码，我们常常使用抽象类来担任 Strategy 的角色，在里面封装公共的代码，因此，在很多应用的场景中，在策略模式中一般会看到模版方法模式的影子。

策略模式的结构

- **封装类：**也叫上下文，对策略进行二次封装，目的是避免高层模块对策略的直接调用。

- **抽象策略：**通常情况下为一个接口，当各个实现类中存在着重复的逻辑时，则使用抽象类来封装这部分公共的代码，此时，策略模式看上去更像是模版方法模式。
- **具体策略：**具体策略角色通常由一组封装了算法的类来担任，这些类之间可以根据需要自由替换。

策略模式代码实现

```
interface IStrategy {
    public void doSomething();
}
class ConcreteStrategy1 implements IStrategy {
    public void doSomething() {
        System.out.println("具体策略 1");
    }
}
class ConcreteStrategy2 implements IStrategy {
    public void doSomething() {
        System.out.println("具体策略 2");
    }
}
class Context {
    private IStrategy strategy;

    public Context(IStrategy strategy){
        this.strategy = strategy;
    }

    public void execute(){
        strategy.doSomething();
    }
}

public class Client {
    public static void main(String[] args){
        Context context;
        System.out.println("-----执行策略 1-----");
        context = new Context(new ConcreteStrategy1());
        context.execute();

        System.out.println("-----执行策略 2-----");
        context = new Context(new ConcreteStrategy2());
        context.execute();
    }
}
```

策略模式的优缺点

策略模式的主要优点有：

- 策略类之间可以自由切换，由于策略类实现自同一个抽象，所以他们之间可以自由切换。
- 易于扩展，增加一个新的策略对策略模式来说非常容易，基本上可以在不改变原有代码的基础上进行扩展。
- 避免使用多重条件，如果不使用策略模式，对于所有的算法，必须使用条件语句进行连接，通过条件判断来决定使用哪一种算法，在上一篇文章中我们已经提到，使用多重条件判断是非常不容易维护的。

策略模式的缺点主要有两个：

- 维护各个策略类会给开发带来额外开销，可能大家在这方面都有经验：一般来说，策略类的数量超过 5 个，就比较令人头疼了。
- 必须对客户端（调用者）暴露所有的策略类，因为使用哪种策略是由客户端来决定的，因此，客户端应该知道有什么策略，并且了解各种策略之间的区别，否则，后果很严重。例如，有一个排序算法的策略模式，提供了快速排序、冒泡排序、选择排序这三种算法，客户端在使用这些算法之前，是不是先要明白这三种算法的适用情况？再比如，客户端要使用一个容器，有链表实现的，也有数组实现的，客户端是不是也要明白链表和数组有什么区别？就这一点来说是有悖于迪米特法则的。

适用场景

做面向对象设计的，对策略模式一定很熟悉，因为它实质上就是面向对象中的继承和多态，在看完策略模式的通用代码后，我想，即使之前从来没有听说过策略模式，在开发过程中也一定使用过它吧？至少在在以下两种情况下，大家可以考虑使用策略模式，

- 几个类的主要逻辑相同，只在部分逻辑的算法和行为上稍有区别的情况。
- 有几种相似的行为，或者说算法，客户端需要动态地决定使用哪一种，那么可以使用策略模式，将这些算法封装起来供客户端调用。

策略模式是一种简单常用的模式，我们在进行开发的时候，会经常有意无意地使用它，一般来说，策略模式不会单独使用，跟模版方法模式、工厂模式等混合使用的情况比较多。

2 用 Java 编程实现二叉树，可以按照值的大小添加节点，前序遍历/中序遍历/后序遍历，按层进行遍历等功能.

首先定义 Node 节点：

```
package tree;
```

```java
public class BTNode {
    private char key;
    private BTNode left, right;

    public BTNode(char key) {
        this(key, null, null);
    }

    public BTNode(char key, BTNode left, BTNode right) {
        this.key = key;
        this.left = left;
        this.right = right;
    }

    public char getKey() {
        return key;
    }

    public void setKey(char key) {
        this.key = key;
    }

    public BTNode getLeft() {
        return left;
    }

    public void setLeft(BTNode left) {
        this.left = left;
    }

    public BTNode getRight() {
        return right;
    }

    public void setRight(BTNode right) {
        this.right = right;
    }
}
```

实现遍历算法：

```java
package tree;

import java.util.LinkedList;
```

```java
import java.util.Stack;

/**
 * 遍历是对树的一种最基本的运算，所谓遍历二叉树，就是按一定的规则和顺序走遍二叉树的所有结点，使每一个结点都被访问一次，而且只被访问一次。
 * 由于二叉树是非线性结构，因此，树的遍历实质上是将二叉树的各个结点转换成为一个线性序列来表示。<br>
 * 设L、D、R分别表示遍历左子树、访问根结点和遍历右子树， 则对一棵二叉树的遍历有三种情况：<br>
 * <li>DLR（称为先根次序遍历）</li>
 * <li>LDR（称为中根次序遍历）</li>
 * <li>LRD（称为后根次序遍历）</li> <br>
 * (1)先序遍历 访问根；按先序遍历左子树；按先序遍历右子树<br>
 * (2)中序遍历 按中序遍历左子树；访问根；按中序遍历右子树<br>
 * (3)后序遍历 按后序遍历左子树；按后序遍历右子树；访问根<br>
 * (4)层次遍历 即按照层次访问，通常用队列来做。访问根，访问子女，再访问子女的子女（越往后的层次越低）（两个子女的级别相同）
 */
public class BTree {
    protected BTNode root;

    public BTree(BTNode root) {
        this.root = root;
    }

    public BTNode getRoot() {
        return root;
    }

    /** 构造树 */
    public static BTNode init() {
        BTNode a = new BTNode('A');
        BTNode b = new BTNode('B', null, a);
        BTNode c = new BTNode('C');
        BTNode d = new BTNode('D', b, c);
        BTNode e = new BTNode('E');
        BTNode f = new BTNode('F', e, null);
        BTNode g = new BTNode('G', null, f);
        BTNode h = new BTNode('H', d, g);
        return h;// root
    }

    /** 访问节点 */
    public static void visit(BTNode p) {
```

```
    System.out.print(p.getKey() + " ");
}

/** 递归实现前序遍历 */
protected static void preorder(BTNode p) {
    if (p != null) {
        visit(p);
        preorder(p.getLeft());
        preorder(p.getRight());
    }
}

/** 递归实现中序遍历 */
protected static void inorder(BTNode p) {
    if (p != null) {
        inorder(p.getLeft());
        visit(p);
        inorder(p.getRight());
    }
}

/** 递归实现后序遍历 */
protected static void postorder(BTNode p) {
    if (p != null) {
        postorder(p.getLeft());
        postorder(p.getRight());
        visit(p);
    }
}

/** 非递归实现前序遍历 */
protected static void iterativePreorder(BTNode p) {
    Stack<BTNode> stack = new Stack<BTNode>();
    if (p != null) {
        stack.push(p);
        while (!stack.empty()) {
            p = stack.pop();
            visit(p);
            if (p.getRight() != null) {
                stack.push(p.getRight());
            }
            if (p.getLeft() != null) {
                stack.push(p.getLeft());
            }
```

```
            }
        }
    }

    /** 非递归实现后序遍历 */
    protected static void iterativePostorder(BTNode p) {
        BTNode q = p;
        Stack<BTNode> stack = new Stack<BTNode>();
        while (p != null) {
            // 左子树入栈
            for (; p.getLeft() != null; p = p.getLeft()) {
                stack.push(p);
            }
            // 当前节点无右子或右子已经输出
            while (p != null && (p.getRight() == null || p.getRight() == q))
{
                visit(p);
                q = p;// 记录上一个已输出节点
                if (stack.empty()) {
                    return;
                }
                p = stack.pop();
            }
            // 处理右子
            stack.push(p);
            p = p.getRight();
        }
    }

    /** 非递归实现中序遍历 */
    protected static void iterativeInorder(BTNode p) {
        Stack<BTNode> stack = new Stack<BTNode>();
        while (p != null) {
            while (p != null) {
                if (p.getRight() != null) {
                    stack.push(p.getRight());// 当前节点右子入栈
                }
                stack.push(p);// 当前节点入栈
                p = p.getLeft();
            }
            p = stack.pop();
            while (!stack.empty() && p.getRight() == null) {// 没有右子树，直接
出栈
                visit(p);
```

```
            p = stack.pop();
        }
        visit(p);//// 有右子树，循环
        if (!stack.empty()) {
            p = stack.pop();
        } else {
            p = null;
        }
    }
}

// level order
public static void levelOrder(BTNode p) {
    if (p == null) {
        return;
    }
    LinkedList<BTNode> queue = new LinkedList<BTNode>();
    queue.add(p);
    while (!queue.isEmpty()) {
        BTNode temp = queue.remove();
        visit(temp);
        if (temp.getLeft() != null) {
            queue.add(temp.getLeft());
        }
        if (temp.getRight() != null) {
            queue.add(temp.getRight());
        }
    }
}

public static void main(String[] args) {
    BTree tree = new BTree(init());
    System.out.print(" Pre-Order:");
    preorder(tree.getRoot());
    System.out.println();
    System.out.print("  In-Order:");
    inorder(tree.getRoot());
    System.out.println();
    System.out.print("Post-Order:");
    postorder(tree.getRoot());
    System.out.println();
    System.out.print(" Pre-Order:");
    iterativePreorder(tree.getRoot());
    System.out.println();
```

```
        System.out.print("  In-Order:");
        iterativeInorder(tree.getRoot());
        System.out.println();
        System.out.print("Post-Order:");
        iterativePostorder(tree.getRoot());
        System.out.println();
        System.out.print("level:");
        levelOrder(tree.getRoot());
        System.out.println();
    }
}
```